CI4251 - Programación Funcional Avanzada
Tarea 1

Simón Castillo
09-10147
<scastb@gmail.com>

Mayo 13, 2013

```
import Data.Functor as F
import Data.Foldable as DF
import Data.Sequence as S
import Data.Map as M
import Data.Monoid
import Data.Maybe
```

1. Fold abstracto

- **(1 punto)** – considere la función `dropWhile` provista en el Preludio y en `Data.List`. Provea una implantación de `dropWhile` empleando el `fold` más apropiado según el caso.

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p xs = fst $ Prelude.foldr f ([],[]) xs
  where f x ~(ys, xs) = (if p x then ys else x:xs, x:xs)
```

  Explicación de la derivación en [**?**] y del uso de $\sim$ en [**?**]

- **(2 puntos)** – Provea una implantación de `foldl` usando `foldr`.

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z xs = Prelude.foldr step id xs z
  where step x g = \a -> g (f a x)
```

  Incluya un diagrama que ilustre el funcionamiento de su implantación.

```
-- foldl (-) 0 [1, 2, 3]
-- = foldr step id [1, 2, 3] 0
-- = step 1 (step 2 (step 3 id)) 0
-- = step 1 (step 2 (\a -> id (a - 3))) 0
-- = step 1 (\b -> (\a -> id (a - 3)) (b - 2)) 0
-- = (\c -> (\b -> (\a -> id (a - 3)) (b - 2)) (c - 1)) 0
-- = (\b -> (\a -> id (a - 3)) (b - 2)) (0 - 1)
-- = (\a -> id (a - 3)) ((0 - 1) - 2)
-- = id (((0 - 1) - 2) - 3)
-- = (((0 - 1) - 2) - 3)
-- = -6
```

  Esta implementación de `foldl` funciona acumulando los cambios sobre una función. Una bonita explicación del funcionamiento de esto como un monoide sobre la composición de funciones está en [**?**].

2. Foldable y Functor

Considere el tipo de datos

```
data Dream b a = Dream a
               | Limbo (b,a)
               | Within a (Seq (Dream b a))
               | Nightmare b
               deriving (Show)
testDream = Within 2 (S.fromList [Dream 4, Limbo (3, 4),
                                  Nightmare 68,
                                  Within 10
                                  (S.fromList [Dream 2, Limbo (4, 3)])
                                 ])
```

- **(1 puntos)** – Construya la instancia `Functor` para el tipo `Dream b`.

```
instance Functor (Dream b) where
  fmap f (Dream x) = Dream (f x)
  fmap f (Limbo (y, x)) = Limbo (y, f x)
  fmap f (Within x y) = Within (f x) (fmap (fmap f) y)
  fmap f (Nightmare y) = (Nightmare y)

asStrings = fmap show testDream
```

- **(2 puntos)** – Construya la instancia `Foldable` para el tipo `Dream b`.

```
instance DF.Foldable (Dream b) where
  foldr f z (Dream x) = f x z
  foldr f z (Limbo (y, x)) = f x z
  foldr f z (Within x y) = f x (DF.foldr g z y)
    where g a b = DF.foldr f b a
  foldr f z (Nightmare y) = z

left = DF.foldl (-) 0 testDream
right = DF.foldr (-) 0 testDream
```

3. Monoid

Considere el tipo de datos `(Data.Map k v)` comentado en clase, que tiene algún tipo de datos `k` como clave y sobre el cual queremos permitir *múltiples valores* asociados a una clave.

Proponga un tipo de datos concreto apoyado en `Data.Map` que permita esa funcionalidad, y entonces:

- **(2 puntos)** – Construya la instancia `Monoid` para este tipo de datos. En la instancia queremos que al combinar dos `Map`, si hay claves repetidas, se *unan* los valores asociados.

```
newtype MultiMap k v = MultiMap (M.Map k (S.Seq v))
                     deriving (Show)
instance Ord k => Monoid (MultiMap k v) where
  mempty = MultiMap M.empty
  (MultiMap a) `mappend` (MultiMap b) = MultiMap $ M.unionWith mappend a b
```

- **(1 punto)** – Escriba un ejemplo de uso con al menos *tres* tablas involucradas, que contengan claves *repetidas* cuyos valores deban combinarse para ejercitar el `Monoid` a la medida.

```
fromMap :: Ord k => M.Map k v -> MultiMap k v
fromMap = MultiMap .  M.map S.singleton
x = fromMap $ M.fromList [("simon", 20), ("daniela", 21),
                          ("squonk", 42)]
y = fromMap $ M.fromList [("simon", -1), ("daniela", 10),
                          ("torvalds", 13)]
w = fromMap $ M.fromList [("simon", 128), ("squonk", 1024)]
z = x `mappend` y `mappend` w
-- z should be:
-- MultiMap (fromList [("daniela",fromList [21,10]),
--                     ("simon",fromList [20,-1,128]),
--                     ("squonk",fromList [42, 1024]),
--                     ("torvalds",fromList [13])])
```

4. Zippers

Considere el tipo de datos

```
data Tree a = Leaf a | Node a (Tree a) (Tree a) (Tree a)
            deriving (Show, Eq)
```

(3 puntos) – diseñe un zipper seguro para el tipo `Tree` proveyendo todas las funciones de soporte que permitan trasladar el foco dentro de la estructura de datos, así como la modificación de cualquier posición dentro de la estructura.

```
data Crumb a = LeftCrumb a (Tree a) (Tree a)
             | CenterCrumb a (Tree a) (Tree a)
             | RightCrumb a (Tree a) (Tree a)
             deriving (Eq, Show)

type Breadcrumbs a = [Crumb a]
type Zipper a = (Tree a, Breadcrumbs a)

focus :: Tree a -> Maybe (Zipper a)
focus t = Just (t, [])

defocus :: Zipper a -> Maybe (Tree a)
defocus (t, _) = Just t

goLeft, goCenter, goRight, goBack :: Zipper a -> Maybe (Zipper a)
goLeft (Node v l c r, bs) = Just $ (l, (LeftCrumb v c r):bs)
goLeft (Leaf _, _) = Nothing

goCenter (Node v l c r, bs) = Just $ (c, (CenterCrumb v l r):bs)
goCenter (Leaf _, _) = Nothing

goRight (Node v l c r, bs) = Just$ (r, (RightCrumb v l c):bs)
goRight (Leaf _, _) = Nothing

goBack (_, []) = Nothing
goBack (t, (LeftCrumb v c r):bs) = Just $ (Node v t c r, bs)
goBack (t, (CenterCrumb v l r):bs) = Just $ (Node v l t r, bs)
goBack (t, (RightCrumb v l c):bs) = Just $ (Node v l c t, bs)

tothetop :: Zipper a -> Maybe (Zipper a)
tothetop (t, []) = Just (t, [])
tothetop z = tothetop $ fromJust $ goBack z

modify :: (a -> a) -> Zipper a -> Maybe (Zipper a)
modify f (Node v l c r, bs) = Just (Node (f v) l c r, bs)
modify f (Leaf v, bs) = Just (Leaf (f v), bs)

testArbol = Node 3
            (Node 2 (Leaf 4) (Leaf 5) (Node 6 (Leaf 7)
                                       (Leaf 8) (Leaf 9)))
            (Leaf 1)
            (Node 10 (Leaf 11) (Leaf 12) (Leaf 13))

movedera = fromJust $ (focus testArbol) >>= goLeft
          >>= goLeft >>= modify (\_ -> 17)
          >>= goBack >>= goBack >>= tothetop >>= goRight
          >>= goRight >>= goBack
```

```
            >>= modify (*2) >>= tothetop >>= goLeft
            >>= goRight >>= goCenter
            >>= goBack >>= goRight >>= tothetop >>= defocus

-- this has to be the coolest thing i have learned in 5 years.
```